6.0 SOFTWARE QUALITY ASSESSMENT

The Software Quality Assessment (SQA) portion of ASP-I presents the results of an investigation of three major areas of software quality: programming conventions, computational efficiency and memory utilization. The following paragraphs discuss the various components that make up a software quality analysis. This is not an exhaustive list, but these sections cover most of the issues that need to be addressed in order to provide an adequate assessment of the code quality. It should be noted, however, that selection of evaluation criteria is a subjective process and that different factors may or may not be applicable to certain models and their implementations. For example, some quality factors may be language and/or compiler dependent or be tied to a specific design requirement not applicable to a class of models.

RADGUNS v.2.0 is a one-on-one simulation of specific antiaircraft artillery systems versus non-reactive aerial targets. In addition to a variety of input files, *RADGUNS* v.2.0 contains 315 subroutines and 84 functions written in FORTRAN. These routines are grouped into major functional units in the following files:

a. RAD1-RAD5 (radar systems)
b. GUN7-GUN130 (AAA artillery systems)
c. RGGUN (routines common to all AAA systems)
d. RGSENSOR (sensor systems other than radar)
e. RGPDET (probability of detection routines)
f. RGECM (countermeasure routines)
g. RGIO (input/output models)
h. RGUTIL (routines common to all weapons systems)
i. RGDIME (ACES/PHOENIX interface)

The goal of this assessment was to evaluate the syntactical construction of these FORTRAN routines. To meet this goal, 27 subroutines and 7 functions were reviewed for adherence to programming standards, algorithm clarity, computational efficiency, and software maintainability. The individual subroutines and evaluations are contained in Appendix A. A list of the categories addressed is shown in Table 6-1. Each module examined was rated on a scale of 0 (poor) to 5 (excellent) for each of the categories listed in the table.

TABLE 6-1. Software Evaluation Criteria.

MOE #1 - Use of Standards:
Criterion #1: Readability
Criterion #2: Modifiability
Criterion #3: ANSI standards
MOE #2 - Programming Conventions:
Criterion #1: Use of comments and headers
Criterion #2: Use of formatted statements
Criterion #3: Logical I/O devices
Criterion #4: Variable declarations
Criterion #5: Variable initialization
Criterion #6: Variable naming conventions
Criterion #7: Algorithm clarity

MOE # 3 - Computational Efficiency:
Criterion #1: Mixed mode calculations
Criterion #2: Use of library functions
Criterion #3: Nested computations
MOE # 4 - Maintainability:
Criterion #1: Portability
Criterion #2: Memory management
Criterion #3: Use of COMMON blocks
Criterion #4: Modularity
Criterion #5: Subroutine tractability

In the course of the evaluation, if a MOE was not applicable to a particular software module, an "N" was placed in that field. An example of this was the "use of formatted statements." Some of the subroutines did not contain any format statements and were therefore not graded against that criterion. For each criterion, a maximum score was calculated based on the total number of modules evaluated multiplied by 5. The absolute sum of the scores given and the percentage of the maximum as well as the average score was tabulated and is presented in Table 6-2.

In general, *RADGUNS* v.2.0 is a well-constructed collection of modular programs which is easy to understand, modify, and maintain. The overall average of all the modules analyzed places the code at 93% of the maximum possible score. The majority of the flaws uncovered were minor in nature (variables not being described in the comments and unused common block variables were two of the most common). More major flaws uncovered included the existence of unused/incomplete code and subroutines that require user modification prior to simulation execution.

TABLE 6-2. Summary of Software Quality Assessment.

Routine	MOE 1.1	MOE 1.2	MOE 1.3	MOE 2.1	MOE 2.2	MOE 2.3	MOE 2.4	MOE 2.5	MOE 2.6	MOE 2.7
MOVANT	3	4	5	4	N	N	4	5	5	4
SIGNL	3	4	5	4	N	N	4	5	5	4
RCVRT	4	4	4	3	4	5	4	5	5	4
ANTTRK	5	5	5	4	N	N	5	5	5	5
SPLGAT	5	5	5	5	N	N	5	5	5	5
RSERVO	5	5	5	4	N	N	5	5	5	5
MTITRK	5	5	5	4	N	N	5	5	5	5
BURST	4	5	5	3	N	N	5	5	5	4
FCCOMP	5	5	5	5	N	N	5	5	5	5
KD	5	5	5	1	N	N	5	5	5	5
NOISE	5	4	5	3	N	N	4	5	5	4
SWEPTA	5	4	5	3	N	N	4	5	5	4
FCCOM1	4	5	5	3	N	N	5	5	5	4
SHOOT	4	5	5	3	5	5	5	5	5	4
RGINP18	5	5	5	5	5	5	5	5	5	5
CHKTRK	4	5	4	4	3	5	5	5	5	4
EVENT	3	5	4	5	4	5	5	5	5	5
PDET	5	5	5	4	N	N	5	5	5	5
DGAM	3	4	5	N	N	5	5	5	5	3
OPLEA1	4	5	5	4	N	N	5	5	5	5
SPDRNG	4	4	4	4	N	N	5	5	5	4
AZDIFF	5	5	5	4	N	N	5	5	5	5
CLUTG	4	5	5	4	3	5	5	5	5	4
HITPRB	4	5	5	5	4	5	5	5	5	5
ORIENT	5	5	5	4	N	N	5	5	5	5
SRCH1	5	5	5	4	4	5	5	5	5	5
LEGDY	5	5	5	4	N	N	5	5	5	5
MYSPEC	5	5	5	4	N	N	5	5	5	5

TABLE 6-2. Summary of Software Quality Assessment. (Contd.)

Routine	MOE 1.1	MOE 1.2	MOE 1.3	MOE 2.1	MOE 2.2	MOE 2.3	MOE 2.4	MOE 2.5	MOE 2.6	MOE 2.7
rgdime.f	3	3	1	3	1	1	1	1	5	3
Count	29	29	29	28	9	10	29	29	29	29
Sum	126	136	137	107	33	46	136	141	145	130
Average	4.3	4.7	4.7	3.8	3.7	4.6	4.7	4.9	5.0	4.5
Max	145	145	145	140	45	50	145	145	145	145
Percent	87%	94%	94%	76%	73%	92%	94%	97%	100%	90%

Module	MOE 3.1	MOE 3.2	MOE 3.3	MOE 4.1	MOE 4.2	MOE 4.3	MOE 4.4	MOE 4.5
MOVANT	5	5	3	5	4	5	4	5
SIGNL	5	5	5	5	5	5	5	5
RCVRT	4	5	5	4	4	5	4	5
ANTTRK	5	5	5	5	5	5	5	5
SPLGAT	5	5	5	5	5	5	5	5
RSERVO	5	5	5	5	5	5	5	5
MTITRK	5	5	N	5	5	5	5	5
BURST	5	5	5	5	5	5	5	5
FCCOMP	N	5	N	5	1	N	5	5
KD	5	5	N	5	5	N	5	5
NOISE	5	5	5	5	4	5	5	5
SWEPTA	5	5	5	5	4	5	5	5
FCCOM1	5	5	5	5	4	5	5	5
SHOOT	5	5	5	5	4	5	5	5
RGINP18	N	5	N	5	5	5	5	5
CHKTRK	N	5	N	5	5	5	5	5
EVENT	N	5	N	5	4	4	5	5
PDET	5	5	5	5	5	5	5	5
DGAM	5	5	5	5	5	N	5	5
OPLEA1	5	5	5	5	5	5	5	5
SPDRNG	5	5	5	5	3	4	5	5
AZDIFF	5	5	N	5	5	5	5	5
CLUTG	5	5	5	5	5	5	5	5
HITPRB	5	5	5	5	5	5	5	5
ORIENT	5	5	N	5	4	4	5	5
SRCH1	5	5	5	5	4	4	5	5
LEGDY	N	5	5	5	4	4	5	5
MYSPEC	5	5	N	5	4	4	5	5
rgdime.f	3	3	3	1	1	1	1	1
Count	24	29	20	29	29	26	29	29
Sum	117	143	96	140	124	120	139	141
Average	4.9	4.9	4.8	4.8	4.3	4.6	4.8	4.9
Max	120	145	100	145	145	130	145	145
Percent	98%	99%	96%	97%	86%	92%	96%	97%

6.1 PROGRAMMING CONVENTIONS

Use of standards is broken into three categories: readability, modifiability, and adherence to ANSI standards. Programming practices which enhance readability include simplicity of program statements and the use of text alignment, spacing, and case.

Programming practices which aid in modifying code include simplicity of program statements and adherence to structured programming techniques which use standard control sequences. ANSI standards are documented in Reference 37 and encompass the form and interpretation of programs expressed in FORTRAN.

ANSI standards are fully maintained in *RADGUNS* v.2.0. Adherence to naming conventions, character positioning, order of statements, and hierarchical structure of logical expressions were seen throughout the code.

6.1.1 Use of Embedded Comments

Comments should not only identify the individual parts of the code (e.g., variable descriptions), but should also describe the functionality and purpose of the tasks being performed by the code. Comments also need to be distributed throughout the code so the user can correlate the comment with the functionality it is describing.

In the *RADGUNS* v.2.0 code examined, most routines were liberally sprinkled with comments. On average, roughly one-third of the lines of code examined constituted comments. Frequently, "blocks" of code are headed up by comments which identify the purpose of the code. This is very helpful to the user. Comments which describe the functionality of the code are less frequent and would be extremely useful to users. As an example, function DGAM contains a single comment which is shown below.

```
C (  INTEGRAL = 1 - (SUM, J = 0 TO N, OF EXP  (J*ALOG(B) - B - ALOG(NFAC
```

This rather terse (and incomplete) comment does tell us we are calculating an integral, but it does not tell us the integral's function. In Reference 38, it is seen that the function calculates the series solution of the incomplete gamma function which is represented as:

$$Integral = 1 - \sum_{j=0}^{N} e^{-B - j\ln(B) - \sum_{i=2}^{N} \ln(i)}$$

To the user, the latter information is much more helpful and should be described within the routine. The reference should also be listed in a comment.

6.1.2 Use of Module Preambles

The majority of modules examined contained detailed preambles which describe the purpose of the module along with a definition of each of the variables used within the module. Ideally, every variable should be included in the preamble list. Often COMMON block variables are omitted from the description.

6.1.3 Source Code Formatting

Simplicity of code is very important. Programmers can often condense several calculations into one line of code, making it difficult to decipher and modify later. Better programming practice is to break out the calculations into simple steps. As an example, suppose we wish to assign a variable y the value of +3 if the variable x is greater than 5, -3 if x is less than -5, and 0 if x is between -5 and 5. One way to accomplish this is the following:

```
y =(((x .lt. -5) .or. (x .gt. 5)) * sign(x) *3 ).
```

A more straightforward approach to coding this task is:

```
if (x .lt. -5 ) then
      y = -3
else if (x .lt. 5) then
      y = 0
else if (x .gt. 5) then
      y = 3
endif
```

Statements frequently come in “groups” or “blocks” which only make sense when taken as a whole. Separation of the “blocks” with blank lines helps direct the reader in much the same way as a period indicates the completion of a sentence. Additionally, indenting of control structures such as if, then, else clauses or do loops can visually illustrate to the reader what is being controlled and how.

Since case is not significant in FORTRAN, it can be used to promote readability. Consistently dedicating upper or lower case text to comments, parameters, and function call statements can assist the reader in distinguishing between these items.

There are a small number of essential control structures which occur over and over again. It is good programming practice to develop a standard approach to representing these structures and to then attempt to apply these representations uniformly to each programming task. In that way, they will be easily recognizable and easily modifiable. The two main types of control structures are iteration (do) loops and if structures. Control structures to avoid are assigned “goto” and assign statements, computed “goto” statements, and arithmetic if statements.

RADGUNS v.2.0 code utilizes the above principles in all subroutines and function routines analyzed. Indeed, the programming example given above is logic which appears in several of the routines (e.g., assigning a gain factor as a function of antenna position). Control structures are indented and standard approaches to representing similar logic is maintained across most routines. Blocks of related statements are separated by blank lines or comment lines and simple, straightforward representations of algorithms are implemented.

6.1.4 Logical File Processing

Algorithm clarity results from the proper application of the items discussed so far: use of good programming standards and conventions, thoughtful visual presentation, simplicity of task representation, and clear informative comments. The routines reviewed in this

assessment satisfied the majority of these goals and were generally easy to read and understand.

Format statements were used in conjunction with input/output statements. They provide information which directs the editing between the internal representation and the character strings of a record or a sequence of records in the file. As mentioned earlier, most of the software modules reviewed did not contain format statements. Of the routines that did, most involve either printing error messages or processing data output. The format statements follow standard ANSI practices and the style of the output is both readable and useful. As this analysis of the code was predominantly syntactical, the completeness of both the type of data and error messages output was beyond the scope of this assessment.

Similar comments can be made regarding the logical I/O statements as those made for the format statements.

The user determines specific model input and output options via formatted parameter files. The user is guided through a series of questions and available options in the selection of specific simulation input and output options. Having a structured set of simulation options helps ensure all necessary parameters get specified and that the user will generate useful and meaningful output.

6.1.5 Variable Declarations

It is good programming practice to declare all variables and identifiers in explicit "type" statements (REAL, INTEGER, etc.), even though the implicit declaration rules of FORTRAN do not require this. In the routines analyzed for *RADGUNS* v.2.0, the programmers typically rely on the implicit declaration rules and only use variable declaration statements when they wish to override the implicit conventions.

Another good programming practice is to explicitly initialize all variables at the start of the subroutine rather than rely on the implicit FORTRAN default values. In *RADGUNS* v.2.0, the programmers explicitly initialize most variables.

The variable naming conventions used in *RADGUNS* v.2.0 are, in general, well thought out. It is usually easy to determine the meaning of a variable or parameter from the name, something which is very useful when trying to understand or modify the code.

6.1.6 Programming Logic

The quality of the programming for *RADGUNS* v.2.0 is generally good. Each subroutine is limited to one specific modeling task. Data is passed in and out through call or common block statements. No hardware dependent extensions or calls or use of unnecessary entry or exit calls was noted in any of the modules examined.

6.2 COMPUTATIONAL EFFICIENCY

This section discusses elements of the code that affect efficient implementation and execution of the software. Computational efficiency is subdivided into three categories: modularity, algorithm development, and variable allocation.

The *RADGUNS* v.2.0 software was not compiled on a variety of platforms and therefore the true portability was not assessed. However, adherence to ANSI standards, explicit declaration and initialization of variables, and modularity of code all go toward ensuring portability of code.

6.2.1 Modularity

RADGUNS v.2.0 models 5 radars and 19 antiaircraft gun systems. Each of these have been modeled as stand-alone software modules. As one might suspect, there is considerable overlap of functionality between each of the radars and each of the gun systems. Subroutines common to each of the radars appear 5 times and subroutines common to each of the gun systems appear 19 times. While the subroutines have identical names, they are not entirely identical and minor modifications customizing them to the particular system exist. While this type of duplication may seem like an inefficient use of memory, the structure allows the user the greatest flexibility and ease of implementation in running the simulation. The one caveat to this would be the incorporation of new functionality. The programmer would need to be aware that additional functionality changes may need to be inserted in multiple locations.

6.2.2 Algorithm Development

As mentioned earlier, the programmer must balance clarity and simplicity with efficient code. While condensing code down to the absolute minimum number of words may be crucial for on-board software where memory space is at a premium, for simulation code run on workstations and requiring substantial user interface, clarity of code is the major driver. *RADGUNS* v.2.0 programmers have recognized this need and have designed the code appropriately. Calculations are kept simple and straightforward. Common library functions are utilized and repetitive code is typically nested via calls to subroutines dedicated to the repetitive task. A majority of the code consists of branching logic based on user-input parameters. The majority of the branching logic is explicitly called out by “if” structures. This type of logic could be condensed to take fewer lines, but much of the clarity to the user would be lost as a result.

6.2.3 Variable Allocation

Variables in *RADGUNS* v.2.0 are allocated either through common block statements or dimension statements.

6.3 MEMORY UTILIZATION

Efficient use of memory by the software has become less important than other quality factors due to its declining cost and increasing availability. *RADGUNS* v.2.0 is not a real-time simulation restricted to limited host platforms. *RADGUNS* v.2.0 programmers implemented proper variable specification and array size declarations throughout the software modules.

6.3.1 Global Memory

COMMON blocks are widely and effectively used in *RADGUNS* v.2.0 code. In a minor number of cases, COMMON block parameters were passed into subroutines which did not use them.

6.3.2 Local Memory

Local variables in *RADGUNS* v.2.0 were found to have correct and sufficient memory allocation. There were rare instances of variables being passed in through the module call statement which were never used within the module.

6.4 IMPLICATIONS FOR MODEL USE

Some improvements in the area of standards for *RADGUNS* v.2.0 include the following:

a. The case and notation of the comments should be standardized. Sometimes the comments are in uppercase, sometimes lowercase, sometimes indented, sometimes separated by rows of asterisks or blank rows, and sometimes directly above the executable code. Having all the comments (for example) left-justified, lowercase, and separated by a blank line would make them stand apart from the executable code, allowing the reader to easily scan through them.

b. All routines should contain a standardized header. Almost all of the routines examined contain an initial set of comments which provide two types of information. First, the comments describe what tasks the routine performs, and second, they provide a descriptive list of the variables, arrays, and constants used within the routine. The level of detail of both types of information varies from routine to routine. Some have no descriptions whatsoever (e.g., function DGAM). Most routines, however, have descriptions for the majority of parameters used within the module. Often common block parameters are not described, and some parameters which were likely deemed to be self-explanatory (e.g., common block constant TWOPI) are omitted from the commented list. While one can usually figure out from the variable name and contextual setting what its purpose is, it is cumbersome to have to track down the one subroutine that contains the definition of interest. All parameters should be included in the descriptive list in each routine in which the parameter is utilized.

c. Variables should be listed in alphabetical order, both in the descriptive comments and in the variable declaration statements. This was not uniformly the case in the routines examined.

d. Centralizing the format statements into one location (typically, just prior to the END statement) assists the reader immensely when a single format statement is used by multiple read/write statements. The majority of the routines examined did not contain format statements, but those that did were inconsistent about their placement within the code. Additionally, the format statement numbers did not always appear in numerically ascending order. Again, this makes it difficult for the reader to track down the location of the statements within the code.

e. Parameter-dependent calls requiring the user to enter the subroutine and modify the code should be eliminated. An example of this was found in subroutine CLUTG. This subroutine computes the power that the radar receives from a ground clutter patch at the specified range, provided the radar beam intersects the earth's surface at that range. Calculation of the surface area intersected is

based on the weapon system. Two calculations are present, and one is commented out while the other is active. If the user-inputs require the non-active area calculation, the user has to go to the source code, toggle the comments, recompile and then execute the simulation. For these cases, logic to select the appropriate calculations should be incorporated into the routine and so be transparent to the user.

f. Unused/incomplete code should be eliminated. The RADGUNS v.2.0 program entitled "rgdime.f" consists of a collection of unfinished subroutines whose function is to add ACES/PHOENIX simulation capabilities. These subroutines are clearly "works in progress" and good configuration management practice should inhibit the inclusion of unfinished code in baseline versions of the code.